Exploring Ada through projectlets

R Srinivasan

Revision 2
March 2023

Contents

Chapter 1

Hello World

1.1 Hello Numbers

In the good old tradition of K&R we will commence our exploration with a hello world. Referring to the program hello, we see the environmental setup in lines 5 - 7. The packages Ada.Text_Io, Ada.Float_Text_Io and Ada.Numerics comprise the predefined language environment in this program.

Listing 1.1: Program hello

```
1  : ------------------------------------------------
2  : --       Program: hello                       --
3  : --       Author: RajaSrinivasan               --
4  : --       Synopsis:                            --
5  : --         Hello world, exploring             --
6  : --         Scientific and Engineering         --
7  : --         applications with Ada.             --
8  : ------------------------------------------------
9  : with Ada.Text_IO;          use Ada.Text_IO;
10 : with Ada.Float_Text_IO; use Ada.Float_Text_IO;
11 : with Ada.Numerics;
12 : procedure hello is
13 : begin
14 :    Put ("Value of pi is =");
15 :    Put (Float (Ada.Numerics.Pi));
16 :    New_Line;
17 :    Put ("And e is =");
18 :    Put (Float (Ada.Numerics.e));
19 :    New_Line;
20 : end hello;
```

Line 11 in this program outputs the constant π converted to the data type float. The constant π is defined in the package Ada.Numerics. The constant as defined does not have a specific data type - hence the conversion to float. If your application needs a higher precision, an appropriate data type might be long_float or long_long_float. In a later section we will explore the attributes of higher precision data types.

Line 6 above specifies the Float_Text_Io which provides the output procedure put utilized in lines 11 and 14. We can infer this since the arguments are of the float data type - the result of converting the constants π and e.

When executed the above program produces the following output on my Ubuntu system:

Listing 1.2: "hello output"

```
Value of pi is = 3.14159E+00
And e is = 2.71828E+00
```

1.2 Higher precision data types

The program hello converted the predefined constants Ada.Numerics.pi to float. If your domain requires a larger range and/or precision than float provides, then long_float could be the solution. However potentially these data types may differ based on the platform. Program hello_attribs illustrates some of the language defined attributes to understand the platform dependencies. Choice of data types could have a significant performance implications - memory footprint as well as computation time.

Listing 1.3: Program hello_attribs

```
 1 : with Ada.Text_IO; use Ada.Text_IO;
 2 : with Ada.Numerics;
 3 :
 4 : with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
 5 : with Ada.Long_Float_Text_IO;    use Ada.Long_Float_Text_IO;
 6 :
 7 : procedure hello_attribs is
 8 : begin
 9 :    Put("Integer'first = "); Put( Integer'First ) ; New_Line ;
10 :    Put("Integer'last = "); Put(Integer'Last) ; New_Line ;
11 :
12 :    Put ("Long_Float'Size = ");
13 :    Put (Long_Float'Size);
14 :    New_Line;
15 :
16 :    Put ("Long_Float'Safe_First = ");
17 :    Put (Long_Float'Safe_First);
18 :    New_Line;
19 :
20 :    Put ("Long_Float'Safe_Last = ");
21 :    Put (Long_Float'Safe_Last);
22 :    New_Line;
23 :
24 :    Put ("Long_Float'Model_Mantissa = ");
25 :    Put (Long_Float'Model_Mantissa);
26 :    New_Line;
27 :
28 :    Put ("Long_Float'Model_Emin = ");
29 :    Put (Long_Float'Model_Emin);
30 :    New_Line;
31 :
32 :    Put ("Long_Float'Model_Epsilon = ");
33 :    Put (Long_Float'Model_Epsilon);
34 :    New_Line;
35 :
36 :    Put ("Long_Float'Model_Small = ");
37 :    Put (Long_Float'Model_Small);
38 :    New_Line;
39 :
40 :    Put ("pi in Long_float = ");
```

```
41 :     Put (Long_Float'Model (Ada.Numerics.Pi));
42 :     New_Line;
43 :
44 :     Put ("e in Long_float = ");
45 :     Put (Long_Float'Model (Ada.Numerics.e));
46 :     New_Line;
47 :
48 : end hello_attribs;
```

On my Ubuntu virtual machine, hello_attribs generates the following output:

Listing 1.4: hello_attribs output

```
Integer'first = -2147483648
Integer'last =  2147483647
Long_Float'Size =           64
Long_Float'Safe_First = -1.79769313486232E+308
Long_Float'Safe_Last =  1.79769313486232E+308
Long_Float'Model_Mantissa =           53
Long_Float'Model_Emin =         -1021
Long_Float'Model_Epsilon =  2.22044604925031E-16
Long_Float'Model_Small =  2.22507385850720E-308
pi in Long_float =  3.14159265358979E+00
e in Long_float =  2.71828182845905E+00
```

As the log indicates Long_float uses 64 bits for storage and values that can be safely be represented as long_float range from -1.79769313486232E+308 to 1.79769313486232E+308.

1.3 Mathematical Functions

The package Ada.Numerics encountered earlier is the root of a comprehensive tree of packages. The program table listed here illustrates the application of several of the functions supported by this predefined language environment.

Listing 1.5: table.adb

```
1 : ------------------------------------------
2 : --       Program: table                 --
3 : --       Author: RajaSrinivasan         --
4 : --       Synopsis:                      --
5 : --         generate a table of values   --
6 : --         of different functions       --
7 : ------------------------------------------
8 : with Ada.Text_IO;  use Ada.Text_IO;
9 : with Ada.Float_Text_IO;  use Ada.Float_Text_IO;
10 : with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
11 : procedure table is
12 :    x     : Float := 0.1;
13 :    x_max : Float := Ada.Numerics.Pi / 2.0;
14 : begin
15 :    Put_Line
16 :      ("  x    | Sqrt(x) | Log(x)   | Log10(x) | exp(x) |       x^3    |       x^x");
17 :    while x <= x_max loop
18 :       Put (x, Aft => 2, Exp => 0); Put ("  |  ");
19 :       Put (Sqrt (x), Aft => 2, Exp => 0); Put ("  |  ");
20 :       Put (Log (x), Exp => 0);  Put ("  |  ");
21 :       Put (Log (x, Base => 10.0), Aft => 2, Exp => 0);  Put ("    |  ");
22 :       Put (Exp (x), Aft => 2, Exp => 0); Put ("  |  ");
23 :       Put ( "**" (x, 3.0), Aft => 7, Exp => 0);  Put ("  |  ");
24 :       Put (x ** x, Aft => 8, Exp => 0);
25 :       New_Line;
26 :       x := x + 0.1;
27 :    end loop;
28 : end table;
```

As used in lines 19-22 the package includes the basic numerical functions such as square root, natural logarithms and so on.

Line 23 illustrates the operator "**" - essentially a function of two variables returning the first variable raised to the power of second variable. Line 24 illustrates an alternate (usually preferred) syntax to use the same operator. This usage requires the use clause for the package where the operator is defined.

Line 18 illustrates a method to tailor the output format. Since the object of the Put is x, we can infer that the procedure is defined in Ada.Float_Text_Io. The Aft and Exp parameters control the number of decimal places and the size of the exponent we want shown in the output.

x	Sqrt(x)	Log(x)	Log10(x)	exp(x)	x^3	x^x
0.10	0.32	−2.30259	−1.00	1.11	0.0010000	0.79432821
0.20	0.45	−1.60944	−0.70	1.22	0.0080000	0.72477967
0.30	0.55	−1.20397	−0.52	1.35	0.0270000	0.69684529
0.40	0.63	−0.91629	−0.40	1.49	0.0640000	0.69314486
0.50	0.71	−0.69315	−0.30	1.65	0.1250000	0.70710677
0.60	0.77	−0.51083	−0.22	1.82	0.2160000	0.73602194
0.70	0.84	−0.35667	−0.15	2.01	0.3430001	0.77905595
0.80	0.89	−0.22314	−0.10	2.23	0.5120001	0.83651167
0.90	0.95	−0.10536	−0.05	2.46	0.7290002	0.90953267
1.00	1.00	0.00000	0.00	2.72	1.0000004	1.00000012
1.10	1.05	0.09531	0.04	3.00	1.3310004	1.11053431
1.20	1.10	0.18232	0.08	3.32	1.7280008	1.24456501
1.30	1.14	0.26236	0.11	3.67	2.1970010	1.40645695
1.40	1.18	0.33647	0.15	4.06	2.7440014	1.60169339
1.50	1.22	0.40547	0.18	4.48	3.3750017	1.83711791

In addition to the functions above, the Elementary_Functions supported include trigonometric and hyperbolic functions.

1.4 Derived Data Types

Real life applications of course manipulate/deal with many different kinds of quantitative items. We have currency numbers, production levels and the like. High integrity systems development demands a disciplined approach to keeping the computations meaningful eliminating nonsensical computations. Ada provides a rich and elegant safety net in its data typing facilities.

The program blood_chem below illustrates some key features.

Listing 1.6: blood_chem.adb

Listing 1.7: blood_chem output

```
Hypoglycemia Range is glucose levels below   7.00000E+01
Hyperglycemia Range is glucose levels above   1.40000E+02
My Glucose Level is   1.40000E+02
My Glucose levels are good
 Assigning good_glucose_value to hypo_value
 blood_chem.adb:57 range check failed
 Assigning hypo_value to hyper_value
 blood_chem.adb:65 range check failed
```

Line 5 declares a new data type Glucose_Concentration_Type. It is based on the fundamental data type float but has a restriction that it has a valid range. Any variable of this type can be assured to have a value in this range. [1]

Lines 6, 8 and 10 also declare data types but these are subtypes. A simplistic way to look at them is as a family of data types which are interchangeable in most computations. Arithmetic and assignments among these data types are allowed while assignment of values of other data types including the base data type float will require explicit fragments of code.

Lines 30, 33 etc illustrates the expressive power of Ada even further. Since the system knows the specifications of a data type, we can ensure that a give value is valid before using it for assignment. Lines 44 and 45 for example perform such an assignment. This is not the violation of the data type rules however. Observe the runtime behaviour of such an assignment.

Lines 21 and 24 introduce new keywords and syntax namely 'First and 'Last. These are language defined attributes applied to data types for example HyperGlycemic_Glucose_Type. As the keyword suggests, the Last attribute provides the highest value that is of the specified data type. Attribute Range in the line 30 refers not to a single value but the range.

1.5 Exercises

Higher precision Convert the above program to display π and e to higher precisions.

Attributes Compare the attributes of the float, long_float and long_long_float data types for your implementation.

Trigonometric functions Generate a table of Sin, Cos and Tan functions for integral angles 1 to 90.

Hyperbolic functions Develop a program to generate a table of hyperbolics sinh, cosh, tanh, sech, cosech and cotanh.

Custom data types Define your own data types for the Cholesterol levels. Different levels for HDL might be Low and High while for the guidelines for LDL might be to think of Borderline ranges.

Custom functions Develop functions to calculate total cholesterol given HDL and the triglyceride level. Wikepedia is a good reference for the relevant metrics.

[1] It is possible to violate this assurance but that will be the exception and takes an explicit command.

Chapter 2

Packages

We have so far used a few packages from the predefined environment. Packages are an elegant means of combining facilities closely related to each other. For example Ada.Numerics.Elementary_Functions provides a number of mathematical functions. In this chapter we will develop our own family of packages.

2.1 Specification

The specification of a package is essentially the visible contract between the user and the package. The services provided are described in the specification with no commitment as to how exactly the services are provided. It is conceivable that given the specification many different distinct implementations may exist.

Package Statistics The package statistics specified below provides a few basic statistical functions. Line 8 declares the name of the package which ends in line 44. Enclosed within these are the specifications of several types, procedures and functions. Also to be noted is the private section starting at line 35 which declares a function.

Type definitions Line 10 declares a new data type - an array of floating point variables. The elements are indexed by natural numbers and of course this is an indefinite array since the array bounds are not specified. Hence the user - when a variable of this array type is declared - will bound the indices.

Procedures Line 16 declares a procedure - which returns more than one result. Arguments to procedures are readable as well as writeable or updatable. The specification of the procedure indicates which arguments are written to (out) and which are updated (in out).

Functions Lines 19 - 34 declare several functions all of which generate a statistic based on an input array. As noted above, the functions return only one value each.

Private parts Line 35 introduces a private section of the package. The declarations of types, functions, procedures etc in a private section of the package are usable in the body of the package or any of the children of the package but not by the users. This is a form of information hiding whereby the users of the package are not bound strongly to the implementation details. Changes in the private section can only affect the body and leaves the users unaffected.

In our example, the private section defines a function to compute central_moment which in turn is used in computing Skewness and Kurtosis. Skewness for example uses the 3rd order central_moment but then it normalizes the moment. Similarly Kurtosis also uses this private function but has to apply a further computation.

Argument data types The type definition samples_type requires particular attention. It is an array of floating point variables and should be a common data type. However the language specifies that the arguments to functions and procedures should be of a named data type. This creates an array type which then is specific to this package. Arguments to the functions in this package can be of this array type and not any other array type. Later chapters will discuss ways of dealing with these situations.

Listing 2.1: Statistics specification

```
1 : ------------------------------------------------
2 : --       Package: statistics                  --
3 : --       Author: RajaSrinivasan               --
4 : --       Synopsis:                            --
5 : --         Specification of the package       --
6 : ------------------------------------------------
7 :
8 : package statistics is
9 :
10 :    type samples_type is
11 :      array (natural range <>)
12 :      of float ;
13 :
14 :    -- minmax computes the minimum and maximum of
15 :    -- the samples
16 :    procedure minmax(samples : samples_type;
17 :                     min_x : out float ;
18 :                     max_x : out float) ;
19 :    function mean(samples : samples_type)
20 :                  return float ;
21 :    function variance(samples : samples_type)
22 :                     return float ;
23 :    function standard_deviation(samples : samples_type)
24 :                                return float ;
25 :
26 :    -- Skewness is the 3rd standardized moment
27 :    -- Ref: https://En.Wikipedia.Org/Wiki/Skewness
28 :    function skewness(samples: samples_type)
29 :                     return float ;
30 :
31 :    -- kurtosis is the 4th standardized moment
32 :    -- Ref: https://en.wikipedia.org/wiki/Kurtosis
33 :    function kurtosis(samples: samples_type)
34 :                     return float ;
35 : private
36 :    -- central_moment computes the central moment
37 :    -- which can be used by the implementations of the
38 :    -- functions skewness and kurtosis.
39 :    -- Since this is private, the function is not
40 :    -- directly accessible by the clients of this package
41 :    function central_moment(samples: samples_type;
42 :                            n : positive )
43 :                           return float ;
44 : end statistics ;
```

2.2 Using the package

Since the specification is the contract with a user, we now turn our attention to the usage of the package statistics. At this stage the implementation details of the procedures and functions of the package are not relevant. The simple test program below simply exercises each of the functions.

Setting up the environment Line 7 declares our intention to use the package - just like the other packages specified in lines 1 - 3. Without a use clause however, any references to facilities in the package would have to be fully specified.

Random Numbers Lines 17 - 21 reference and use the facilities of another library package - to generate an array of uniformly distributed random numbers.

Line 30 invokes the procedure to get the minimum and maximum of the array Heights. Lines 36, 40, 48 and 52 invoke functions in the same package each returning a value.

Listing 2.2: Test Program 1

```
 1 : with Ada.Text_Io; use Ada.Text_Io;
 2 : with Ada.Integer_Text_Io; use Ada.Integer_Text_Io ;
 3 : with Ada.Float_Text_Io; use Ada.Float_Text_Io ;
 4 :
 5 : with Ada.Numerics.Float_Random ;
 6 :
 7 : with statistics ;
 8 :
 9 : procedure test1 is
10 :    gen : Ada.Numerics.Float_Random.Generator ;
11 :    NUMBER_OF_SAMPLES : constant := 10000 ;
12 :    MAX_HEIGHT : constant Float := 200.0 ;
13 :    heights : statistics.samples_type( 1..NUMBER_OF_SAMPLES ) ;
14 :    min_x, max_x : float ;
15 : begin
16 :
17 :    Put_line("Random sample data");
18 :    Put("Number of Samples =");
19 :    Put(NUMBER_OF_SAMPLES) ;
20 :    Put(" Max Height = ");
21 :    Put( MAX_HEIGHT ) ;
22 :    New_Line ;
23 :
24 :    for idx in 1..NUMBER_OF_SAMPLES
25 :    loop
26 :       heights( idx ) := MAX_HEIGHT *
27 :                   Ada.Numerics.Float_Random.Random(gen) ;
28 :    end loop ;
29 :
30 :
31 :    statistics.minmax( heights , min_x , max_x ) ;
32 :    put("Min ="); put(min_x) ;
33 :    put("; Max ="); put(max_x) ;
34 :    new_line ;
35 :
36 :    put("Mean = ");
```

```
37 :     put(statistics.mean(heights)) ;
38 :     new_line ;
39 :
40 :     put("Variance = ") ;
41 :     put(statistics.variance(heights)) ;
42 :     new_line ;
43 :
44 :     put("Std Deviation = ");
45 :     put(statistics.standard_deviation(heights)) ;
46 :     new_line ;
47 :
48 :     put("Skewness = ");
49 :     put(statistics.skewness(heights)) ;
50 :     new_line ;
51 :
52 :     put("Kurtosis = ");
53 :     put(statistics.kurtosis(heights)) ;
54 :     new_line ;
55 :
56 : end test1 ;
```

Listing 2.3: Sample Output of test1

```
Random sample data
Number of Samples =       10000 Max Height =  2.00000E+02
Min = 5.46962E-02; Max = 1.99986E+02
Mean =  1.00511E+02
Variance =  3.34732E+03
Std Deviation =  5.78560E+01
Skewness = -2.08577E-02
Kurtosis =  1.79570E+00
```

2.3 Implementation or Body

The test program in the previous section of course did not have to concern itself with how the functions e.g. kurtosis is actually implemented. Before the executable of the test program is built, the implementation or body needs to be completed.

An organization for example could draw up the specification for a package and have different competing vendors provide distinct implementations. An instructor may provide such a specification and have each student furnish their own implementation. The instructor can also conceivably provide the test program producing hopefully identical results albeit with different bodies. As long as the implementations conform to the specification in syntax and semantics, the program can meet its goals.

Listing 2.4: Statistics implementation

```
1 : ------------------------------------------
2 : --       Package: statistics           --
3 : --       Author: RajaSrinivasan        --
4 : --       Synopsis:                     --
5 : --         Body (implementation) of the --
6 : --         package                     --
7 : ------------------------------------------
8 :
```

```
 9 : with Ada.Numerics.Elementary_Functions ;
10 :
11 : package body statistics is
12 :    procedure minmax(samples : samples_type;
13 :                     min_x : out float ;
14 :                     max_x : out float)  is
15 :       first : integer := samples'first ;
16 :       pairs : integer := samples'length / 2 ;
17 :    begin
18 :       min_x := samples( first ) ;
19 :       max_x := samples( first ) ;
20 :       if samples'length rem 2 = 1
21 :       then
22 :          -- odd number of elements
23 :          first := first + 1 ;
24 :       end if ;
25 :       for idx in first .. first + pairs - 1
26 :       loop
27 :          if samples(idx) > samples(idx+pairs)
28 :          then
29 :             if samples(idx) > max_x
30 :             then
31 :                max_x := samples(idx) ;
32 :             end if ;
33 :             if (samples(idx+pairs)) < min_x
34 :             then
35 :                min_x := samples(idx+pairs) ;
36 :             end if ;
37 :          else
38 :             if samples(idx) < min_x
39 :             then
40 :                min_x := samples(idx) ;
```

Line 11 indicates that this is the package body or the implementation of the package.

At lines 12 is the implementation of the procedure minmax - truncated here for brevity. The complete body of course has to include the impelementations of all other functions including the private function central_moment.

Studying this carefully one will notice that there are no assumptions made about the samples array bounds. C programmers tend to think of array indices in the range 0 to n-1 while Fortran (or Matlab) programmers might be more comfortable with the bounds 1 to n. Ada enables and the function minmax makes full use of this fact by using the attributes 'First, 'Last, 'Length and so on. The attributes are applied to variables to obtain the actual arguments' attributes.

The algorithm itself for minmax appears to be different from the more obvious simpler alternative. It is completely valid to replace this implementation with a simpler or a faster more complex algorithm.

2.4 SubPackages

A package like statistics can potentially be the root of a family of packages. The following listing specifies a subpackage dependance of statistics. Such a subpackage can potentially specify its own data types, functions and procedures.

Line 2 declares a function. Its parameters are of a data type defined in the root package namely statistics. Because this is a subpackage, all the declarations of the parent package are automatically inherited.

Listing 2.5: SubPackage Dependance

```
1 : package statistics.dependance is
2 :    function Covariance( x : samples_type ;
3 :                         y : samples_type )
4 :      return float ;
5 :    function Correlation_Coefficient( x : samples_type ;
6 :                                      y : samples_type )
7 :                                    return float ;
8 :
9 : end statistics.dependance ;
```

Line 5 of the following test program includes the above subpackage in its environment. Even though the parent package statistics is not explicitly specified, the declarations are implicitly visible.

Lines 16 and 17 declare the variables of the data type declared in the parent package.

Line 33 generates uniformly distributed random numbers which is used to make up the test data.

Listing 2.6: Dependance test program

```
1 : with Ada.Text_Io; use Ada.Text_Io;
2 : with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3 : with Ada.Float_Text_Io; use Ada.Float_Text_Io ;
4 : with Ada.Numerics.Float_Random ;
5 :
6 : with statistics.dependance ;
7 :
8 : procedure test2 is
9 :    gen : Ada.Numerics.Float_Random.Generator ;
10 :
11 :    NUMBER_OF_PAIRS : constant := 10000 ;
12 :
13 :    MIN_HEIGHT : constant Float := 500.0 ;  -- centimeters
14 :    MAX_DIFF : constant Float := 1000.0 ; -- centimeters
15 :
16 :    heights_father : statistics.samples_type( 1..NUMBER_OF_PAIRS) ;
17 :    heights_son : statistics.samples_type(1..NUMBER_OF_PAIRS) ;
18 : begin
19 :    Put_line("Random sample data");
20 :    Put("Number of Samples =");
21 :    Put(NUMBER_OF_PAIRS) ;
22 :    New_Line ;
23 :    Put(" Min Height = ");
24 :    Put( MIN_HEIGHT ) ;
25 :    Put(" Max difference = ");
26 :    Put( MAX_DIFF ) ;
27 :    New_Line ;
28 :
29 :    for idx in 1..NUMBER_OF_PAIRS
```

```
30 :     loop
31 :        heights_father( idx ) := MIN_HEIGHT +
32 :                                 MAX_DIFF *
33 :                                 Ada.Numerics.Float_Random.Random(gen) ;
34 :        heights_son( idx ) := Heights_Father( Idx ) / 2.0 ;
35 :
36 :     end loop ;
37 :
38 :     put("Covariance is =");
39 :     put(statistics.dependance.covariance(heights_father, heights_son)) ;
40 :     New_Line ;
41 :
42 :     put("Correlation Coefficient is =");
43 :     put(statistics.dependance.correlation_coefficient(heights_father, heights_so
44 :     new_line ;
45 : end test2 ;
```

Error handling The body listed below introduces in line 13 a possible way to handle errors. In this statement, an exception Program_Error is raised when one of the preconditions is not met. Another way might have been to return a status but then we can no longer call it a function. The exception in this case is a language defined exception. Later chapters will introduce custom defined exceptions. Handling exceptions will also be addressed later.

Also of interest to note the lines 15 and 16 illustrating the visibility of all the functions and procedures of the parent package in the subpackage.

Listing 2.7: Body of dependance

```
 1 : package body statistics.dependance is
 2 :
 3 :    function Covariance( x : samples_type ;
 4 :                         y : samples_type )
 5 :                        return float is
 6 :       mean_x,
 7 :       mean_y : float ;
 8 :       cov_xy : float := 0.0 ;
 9 :       exp_x, exp_y : float := 0.0 ;
10 :    begin
11 :       if x'length /= y'length
12 :       then
13 :          raise Program_Error ;
14 :       end if ;
15 :       mean_x := mean(x) ;
16 :       mean_y := mean(y) ;
17 :
18 :       for x_idx in x'range
19 :       loop
20 :          exp_x := (x(x_idx) - mean_x) ;
21 :          exp_y := (y(x_idx) - mean_y) ;
22 :          cov_xy := cov_xy + exp_x * exp_y ;
23 :       end loop ;
24 :       cov_xy := cov_xy / float(x'length) ;
25 :       return cov_xy ;
26 :    end Covariance ;
27 :
```

```
28 :     function Correlation_Coefficient( x : samples_type ;
29 :                                       y : samples_type )
30 :                                     return float is
31 :        stddev_x,
32 :        stddev_y : float ;
33 :        cov_xy : float ;
34 :     begin
35 :        cov_xy := covariance(x,y) ;
36 :        stddev_x := standard_deviation(x) ;
37 :        stddev_y := standard_deviation(y) ;
38 :        return cov_xy / (stddev_x * stddev_y) ;
39 :     end Correlation_Coefficient ;
40 : end statistics.dependance ;
```

The package statistics.random listed below is another subpackage of statistics. Unlike the package dependance however, there is no reference to any new data types or procedures defined in the parent. The goal is just to establish a namespace and collect all functionality related to statistics in this tree.

Listing 2.8: Specification of subpackage random

```
1 : package statistics.random is
2 :     -- Function: Weibull
3 :     -- Return a random number with a Weibull distribution
4 :     function Weibull( scale : float ;
5 :                       shape : float )
6 :       return float ;
7 : end statistics.random ;
```

2.5 Exercises

Different implementation Replace the minmax algorithm with a simpler implementation reducing the number of lines of code.

Median Add a function to return the median of the array.

Histogram Add a procedure to the package and modify the test program to use and print the results to the following specification:

```
type histogram_type is array (integer range <>) of integer ;
-- computes the histogram of the samples
-- the dimensions of the argument bins specify the bin size
procedure histogram( samples : samples_type ;
                     bins : out histogram_type ) ;
```

Randome variables - normal distribution The examples for this chapter includes a subpackage statistics.random. Add functions to this package to generate random numbers in different distributions e.g. Normal and chi.

Chapter 3

Unit Tests

3.1 Background

In this chapter, some of the unit testing infrastructure available with Ada is introduced. The importance of unit testing in the development of large and complex applications can hardly be over emphasized. Scientific and engineering applications in particular have to incorporate unit testing in the development life cycle.

AUnit Based on the JUnit framework for Java and other such frameworks, AUnit is one of several unit testing frameworks. Though this is not part of the Ada language specifications, AUnit is likely to be installed (or easily installed) on most common development platforms.

3.2 Unit testing Complex Numbers

In order to illustrate the unit testing infrastructure, we explore the predefined package Ada.Numerics.Complex_Types. Complex numbers are a composite data type based on the fundamental data types supported by Ada by means of language specified packages.

Unit tests for the package Ada.Numerics.Complex_Types are defined in the package complex_tests. In the following specification, line defines a new test case data type Complex_Test. This is a type derived from the base type Test_Case. This is the first time we run into object orientation, dispatching and other such notions as implemented in Ada. Later chapters will discuss these in some detail. At this stage suffice it to note that Register_Tests and Name are procedure and function that are required to be defined for each Test_Case.

Lines 10 - 12 define the functions that are specific to the task at hand ie testing the package Ada.Numerics.Complex_Types.

Listing 3.1: Specification of complex_tests

```
 1 : with AUnit ; use AUnit ;
 2 : with AUnit.Test_Cases ; use AUnit.Test_Cases ;
 3 :
 4 : package Complex_Tests is
 5 :
 6 :     type Complex_Test is new Test_Cases.Test_Case with null record ;
 7 :     procedure Register_Tests( T : in out Complex_Test );
 8 :     function Name( T : Complex_Test ) return Message_String ;
 9 :
10 :     procedure Test_Abs( T : in out Test_Cases.Test_Case'Class ) ;
11 :     procedure Test_Argument( T : in out Test_Cases.Test_Case'Class ) ;
12 :     procedure Test_Polar( T : in out Test_Cases.Test_Case'Class ) ;
13 :
```

```
14 : end Complex_Tests ;
```

Reviewing the implementation of the package, line 12 starts the implementation of the procedure Register_Tests which registers the 3 test routines in the lines 15 - 17. Implementation of the function Name is pretty straightforward.

Each test routine tests a distinct feature of the package comparing expected results and the actual results. If they do not match then the Assert procedure prints an appropriate message and the test case concludes.

Listing 3.2: Implementation of complex_tests

```
1  : with Ada.Numerics.Complex_Types ;
2  : with Ada.Numerics.Elementary_Functions ;
3  :
4  : with Ada.Text_Io ; use Ada.Text_Io ;
5  : with Ada.Complex_Text_Io ;
6  : with Ada.Float_Text_Io ; use Ada.Float_Text_Io ;
7  :
8  : with AUnit.Assertions ; use AUnit.Assertions ;
9  :
10 : package body complex_tests is
11 :
12 :     procedure Register_Tests( T : in out Complex_Test ) is
13 :        use AUnit.Test_Cases.Registration ;
14 :     begin
15 :         Register_Routine( T , Test_Abs'access , "Test Absolute ") ;
16 :         Register_Routine( T , Test_Argument'access , "Test Argument");
17 :         Register_Routine( T , Test_Polar'access , "Test Polar") ;
18 :     end Register_Tests ;
19 :
20 :     function Name( T : Complex_Test ) return Message_String is
21 :     begin
22 :         return Format("Complex Tests") ;
23 :     end Name ;
24 :
25 :     procedure Test_Abs( T : in out Test_Cases.Test_Case'Class ) is
26 :         C1 : Ada.Numerics.Complex_Types.Complex := ( Re => 1.0 ,
27 :                                 Im => 1.0 ) ;
28 :     begin
29 :         AUnit.Assertions.Assert(
30 :                 Ada.Numerics.Complex_Types.Modulus(C1) =
31 :                 Ada.Numerics.Elementary_Functions.Sqrt(2.0)
32 :               , "Absolute value of 1.0 + j");
33 :     end Test_Abs ;
```

```
34 :
35 :      procedure Test_Argument( T : in out Test_Cases.Test_Case'Class ) is
36 :         C1 : Ada.Numerics.Complex_Types.Complex := ( Re => 1.0 ,
37 :                                                      Im => 1.0 ) ;
38 :      begin
39 :          AUnit.Assertions.Assert(
40 :             Ada.Numerics.Complex_Types.Argument(C1) =
41 :             Ada.Numerics.Elementary_Functions.Arctan(1.0)
42 :           , "Argument of 1.0 + j");
43 :      end Test_Argument ;
44 :
45 :      procedure Test_Polar( T : in out Test_Cases.Test_Case'Class ) is
46 :         use type Ada.Numerics.Complex_Types.Complex ;
47 :         C1 : Ada.Numerics.Complex_Types.Complex := ( Re => 1.0 ,
48 :                                                      Im => 1.0 ) ;
49 :         C2 : Ada.Numerics.Complex_Types.Complex ;
50 :         C3 : Ada.Numerics.Complex_Types.Complex ;
51 :         EPSILON : Float := 0.1e-5 ;
52 :      begin
53 :         C2 := Ada.Numerics.Complex_Types.Compose_From_Polar(
54 :                     Ada.Numerics.Elementary_Functions.Sqrt(2.0) ,
55 :                     Ada.Numerics.Pi / 4.0 ) ;
56 :          C3 := C2 - C1 ;
57 :          Put("C1 ") ;
58 :          Ada.Complex_Text_Io.Put(C1) ;
59 :          Put(" C2 ");
60 :          Ada.Complex_Text_Io.Put(C2) ;
61 :          Put(" C3 ");
62 :          Ada.Complex_Text_Io.Put(C3) ;
63 :          new_line ;
64 :          AUnit.Assertions.Assert(
65 :             Ada.Numerics.Complex_Types.Modulus(C3) < EPSILON and
66 :             Ada.Numerics.Complex_Types.Argument(C3)  < EPSILON
67 :           , "Polar representation of 1.0 + j. difference < EPSILON");
```

```
68 :            AUnit.Assertions.Assert(
69 :               C1 = C2
70 :             , "Polar representation of 1.0 + j; Equality");
71 :      end Test_Polar ;
72 :
73 : end complex_tests ;
```

3.3 Test suite

One or more test cases can be combined to form a test suite. In the body of the suite listed below, line 4 declares an object of the type complex_test encountered earlier. Lines 9 and 10 registers the test case in the test suite after registering the test steps by invoking the Register_Tests procedure.

Listing 3.3: Test suite body

```
 1 : with complex_tests ;
 2 :
 3 : package body complex_suite is
 4 :     t : aliased Complex_Tests.Complex_Test ;
 5 :     function suite return Access_Test_Suite is
 6 :         Ret : constant Access_Test_Suite := new Test_Suite;
 7 :
 8 :     begin
 9 :         Complex_Tests.Register_Tests( T );
10 :         Ret.Add_Test( T'access );
11 :         return Ret ;
12 :     end suite ;
13 : end complex_suite ;
```

At this stage the test suite is complete and is ready for invocation. The main program complex_unit_test listed below, establishes a runner in line 8 and a reporter in line 9. Line 11 sets the series in motion and the unit tests get called and the results reported through the reporter.

Reporting the results In the above example, the reporter is a Text_Reporter. Other forms of reports e.g. in xml or JSON can be visualized and indeed the framework supports an xml reporter.

Listing 3.4: Unit Test Driver

```
 1 : with Complex_Tests ;
 2 : with Complex_Suite ;
 3 : with AUnit.Test_Suites ; use AUnit.Test_Suites ;
 4 : with AUnit.Run ;
 5 : with AUnit.Reporter.Text ;
 6 :
 7 : procedure Complex_Unit_Test is
 8 :     procedure Run is new AUnit.Run.Test_Runner( Complex_Suite.Suite );
 9 :     Reporter : AUnit.Reporter.Text.Text_Reporter ;
10 : begin
11 :     Run(Reporter);
12 : end Complex_Unit_Test ;
```

Listing 3.5: Execution log

```
C1 ( 1.00000E+00, 1.00000E+00) C2 ( 1.00000E+00, 1.00000E+00) C3 (-5.96046E-08,-5.96046E-08)

OK Complex Tests : Test Absolute
OK Complex Tests : Test Argument

FAIL Complex Tests : Test Polar
     Polar representation of 1.0 + j; Equality
     at complex_tests.adb:68

Total Tests Run:    3
Successful Tests:   2
Failed Assertions:  1
Unexpected Errors:  0
```

3.4 Exercises

Investigate the Unit Test failure Investigate the failure reported in the unit test and fix

Add a test case Add another test case Complex_Arithmetic_Test to test the basic arithmetic functions of complex variables and incorporate in the test suite.

Convert to xml reports Convert the unit test to generate xml report instead of just a text report.

Chapter 4

Data Structures

In this chapter we explore further ways of thinking about data items and organizing them. Along the way we review more of the predefined language packages.

4.1 Projectlet markov

The projectlet markov models the changes in a population over a number of cycles. A person can have a preferance for 1 out of 3 possible desserts. Over a period, the preferance may transition to a different dessert. The transition probabilities can be specified as a matrix, each row representing the probability of transition in a cycle. We start with a population and some number of persons classified by their preferences. Over a period, the preferance profile of the population changes which we will study in this projectlet.

4.1.1 Matrices

The transition probabilities as shown below will be read from a data file for the implementation of the markov model.

The subpackage probability below introduces the declaration of a matrix in lines 3 - 5. The two indices are each in the range 0..n where n is the natural'last. The actual size a of a matrix then is specified by the user of the package. Each element of this array is of type probability_type declared in line 2.

Lines 6 and 10 declare convenience functions Valid and Show. Such functions may or may not be used by a user of the package but can be very useful diagnostic tools. For example the function Valid can review a transition matrix to ensure for example that the probabilities in each row sum up to 1.0.

Line 8 declares the procedure load which loads the data from a file supplied to create a transition matrix. Since the transition matrix type specifies only that the indices are natural numbers, the exact ranges of these indices are determined by the user package. In other words, the procedure load should be able to handle any size of matrix.

Listing 4.1: Probability Specification

```
1 : package statistics.probability is
2 :    type probability_type is new float range 0.0 .. 1.0 ;
3 :    type transition_matrix_type is
4 :      array (natural range <> ,
5 :              natural range <>) of probability_type ;
6 :    function Valid(mat : transition_matrix_type)
7 :                    return boolean ;
```

```
 8 :     procedure load(filename : string ;
 9 :                          mat : out transition_matrix_type ) ;
10 :     procedure show(mat : transition_matrix_type ) ;
11 : end statistics.probability ;
```

The program markov below is the user of the probability package and implements the simulation that we set out to do. Line 7 declares the transition matrix with the limits being 1..num_states. num_states being a variable suggests correctly that the array limits can be variables - as long they have appropriate values at the time the declaration of the array is encountered.

Lines 9 & 10 declare and initialize the population as a vector of persons with a count for each of the 3 states.

Line 26 relies on the load procedure to fill up the transition matrix from the file tmatrix.dat. It is the user's responsibility to ensure that the file has data items matching the matrix. Similarly the data items may be expected to be compatible with the probability_type - the data type of the matrix elements.

Lines 9 & 10 declare an array to store the number of persons presumably in one of 3 states.

Listing 4.2: Markov chain application

```
 1 : with Ada.Integer_Text_IO ; use Ada.Integer_Text_IO ;
 2 : with Ada.Text_IO; use Ada.Text_IO ;
 3 :
 4 : with statistics.probability ;
 5 : procedure markov is
 6 :    num_states : integer := 3 ;
 7 :    transitions : statistics.probability.transition_matrix_type( 1..num_states , 1 .. num_states ) ;
 8 :    type Persons_type is  array (1..num_states) of integer ;
 9 :    persons : Persons_Type
10 :      := ( 1200 , 1300 , 500 ) ;
11 :
12 :    procedure cycle is
13 :       next_cycle : Persons_Type := (others => 0) ;
14 :    begin
15 :       for statefrom in 1..num_states
16 :       loop
17 :          for statenext in 1..num_states
18 :          loop
19 :             next_cycle( statenext ) :=
20 :               integer( float(persons(statefrom)) *
21 :                        float(transitions(statefrom,statenext)) +
22 :                        float(next_cycle(statenext)))  ;
23 :          end loop ;
24 :       end loop ;
25 :       persons := next_cycle ;
26 :    end cycle ;
27 : begin
28 :    statistics.probability.load( "tmatrix.dat" , transitions ) ;
29 :    statistics.probability.show( transitions ) ;
30 :    for cyc in 1..20
31 :    loop
32 :       cycle ;
33 :       if cyc rem 2 = 0
```

```
34 :         then
35 :            put("After␣"); put(cyc) ; put("␣cycles␣:␣");
36 :            for p in persons'range
37 :              loop
38 :                 put( persons(p) ) ; put ("␣") ;
39 :              end loop ;
40 :            new_line ;
41 :         end if ;
42 :      end loop ;
43 : end markov ;
```

4.1.2 Internal procedures

We encountered procedures being part of packages in the previous chapter. Lines 12 - 24 here declare a local procedure. The procedure is invoked in line 30. Since the procedure has no parameters, the invocation requires only the name.

Line 31 introduces the rem operator to identify even numbers. Every even cycle the program prints out the current state of the population.

4.1.3 Matrix Attributes

The package body below illustrates several nifty features The procedure load starts at line 32. It is expected to read the input file identified by filename and fill up the elements of mat. Note that the dimensions of mat are not passed in explicitly.

Line 38 and 39 query the Ada runtime for the dimensions of mat. Since mat is 2 dimensional 'first(1) returns the low limit of the first index. Similarly 'first(2) returns the low limit of the second index. Analogously line 44 obtains the upper limit of the second index. Other such attributes like 'range can also be used. This allows the user program to decide the appropriate ranges. C programmers might be comfortable using 0..n-1 while fortran programmers might prefer to think in terms of 1..n - both of which are supported.

Listing 4.3: Probability Body

```
1 : with Ada.Text_IO ; use Ada.Text_IO ;
2 :
3 : package body statistics.probability is
4 :    package Probability_Text_Io is new
5 :      Ada.Text_IO.Float_IO( probability_type ) ;
6 :    use Probability_Text_Io ;
7 :    tolerance : constant probability_type := 1.0e-3 ;
8 :    function Valid(mat : transition_matrix_type) return boolean is
9 :       prob : probability_type ;
10 :   begin
11 :      if mat'length(1) /= mat'length(2)
12 :      then
13 :         raise Program_Error ;
14 :      end if ;
15 :
16 :      for row in mat'range(1)
17 :      loop
18 :         prob := probability_type'first ;
19 :         for col in mat'range(2)
20 :         loop
21 :            prob := prob + mat(row,col) ;
22 :         end loop ;
23 :         if abs(prob - probability_type'last) > tolerance
24 :         then
25 :            return false ;
26 :         end if ;
27 :      end loop ;
28 :      return true ;
29 :   exception
30 :         when others => return false ;
31 :   end Valid ;
32 :   procedure load(filename : string ;
```

```
33 :                        mat : out transition_matrix_type ) is
34 :        input_file : file_type ;
35 :        row,col : natural ;
36 :     begin
37 :        open(input_file , in_file , filename ) ;
38 :        row := mat'first(1) ;
39 :        col := mat'first(2) ;
40 :        while not end_of_file(input_file)
41 :        loop
42 :           get( item => mat(row,col) , file => input_file ) ;
43 :           col := col + 1 ;
44 :           if col > mat'last(2)
45 :           then
46 :              row := row + 1 ;
47 :              col := mat'first(2) ;
48 :           end if ;
49 :        end loop ;
50 :        close(input_file) ;
51 :     end load ;
52 :     procedure show(mat : transition_matrix_type ) is
53 :     begin
54 :        for r in mat'range(1)
55 :        loop
56 :           for c in mat'range(2)
57 :           loop
58 :              put( mat(r,c) ) ;
59 :              put ( " " ) ;
60 :           end loop ;
61 :           new_line ;
62 :        end loop ;
63 :     end show ;
64 : end statistics.probability ;
```

4.1.4 Package Instantiation

Line 4 in the body above creates a new package or instantiates a library package in this case Ada.Text_Io.Float_Io but customized for the probability_type. (The notion of such generic packages and functions will be explored in a later chapter.) This is required since probability_type has been created as a new data type albeit based on the float data type. This package Probability_Text_Io then can read values that can only be of probability_type from text files.

What happens when an item read from a file fails to meet the conditions of the appropriate type e.g. 12.0 in this case? We can expect such a condition to be handled by raising an exception.

4.1.5 Text File Input

The packages Ada.Text_Io and its subpackages collectively support reading from text files. The text file is opened in line 17 and closed in line 50. Line 40 illustrates an important function End_Of_File which returns whether the end of file marker has been reached or not. Line 42 retrieves an item from the input file. Line 42 also illustrates the named association of parameters. In line 37 where there were 3 parameters, without a named association, we took advantage of the ordered association. The first argument is a file, the third the file name and so on.

Since the get in line 42 attempts to retrieve a probability value, it requires a customized package for probability_type. We created just such a package in line 4. Also since line 5 includes a use clause, line 42 did not have to explicitly state the origin of the procedure. Without the use clause of course, we can use the complete reference Probability_Text_Io.get.

Similarly line 58 where the probability value is being displayed using the put procedure, the complete reference Probability_Text_Io.put could have been used instead of the use clause in line 5.

4.1.6 Pitfalls of floating point computations

The population transition simulation log shows that after 14 cycles, the population appears to have converged to a preferance profile. However the population has also grown in size - which was not intended. Careful analysis may be required to establish the root cause of this failure. As a starting point the state transition as implemented in lines 19 - 22 should be investigated.

Floating point computational difficulties are not unique to Ada nor does Ada obviate the need for understanding the underlying architecture and a deliberate approach to dealing with them.

Listing 4.4: Population transition simulation results

```
 3.00000E-01   3.00000E-01   4.00000E-01
 2.00000E-01   6.00000E-01   2.00000E-01
 1.00000E-01   1.00000E-01   8.00000E-01
After          2 cycles :          553          1029          1418
After          4 cycles :          495           865          1640
After          6 cycles :          480           813          1706
After          8 cycles :          476           797          1728
After         10 cycles :          475           792          1734
After         12 cycles :          474           791          1736
After         14 cycles :          474           791          1738
After         16 cycles :          474           791          1738
After         18 cycles :          474           791          1738
After         20 cycles :          474           791          1738
```

4.1.7 Exercises

Convergence At run time, how many iterations in the program markov are required for the transitions to converge? Transitions are said to converge when the number of persons in each state does not change each cycle. Modify the program to stop upon convergence.

Floating point computations Upon convergence do the number of persons add up to the expected number? What could be the source of this error? Correct this program for this problem.

Robustness The robustness of the load procedure in statistics.probability is highly questionable. Among the errors that are not handled adequately are if the file does not contain the required number of items (or more items than required). While the procedure functions correctly if the items all conform to the probability_type, what happens if this is not the case? Improve the implementation and test again with the same data file.

4.2 Projectlet foodsdb

In this projectlet we introduce the record datastructure which is similar to all other languages; however the notion of equality of records can be richer than a low level memory comparison and is illustrated by an implementation of the equality operator. In addition the language defined containers are introduced.

4.2.1 Records

The record data structure supports a heterogeneous collection named items. The specification of the foods package below declares a record in lines 12 - 18. The name is food_item_type and it contains the components name, carbs, calories and glycemic_index_type. Each of the components of course is of a different data type, name being of a predefined data type Unbounded_String from Ada.Strings.Unbounded while the data types of other components are declared locally in lines 7 - 10.

4.2.2 Fixed Point

Line 7 declaring carb_content_type introduces a new fundamental data type whereby numbers are not continuous but discrete with a delta of 0.1. This means that 0.1 is the minimum difference between any two variables of this data type. The range specification says the values may be between 0.0 and 250.0. In this data type, there only 10 unique numbers between 0.0 and 0.9 as well as between 249.1 and 250.0.

Though fixed point types do support decimals eg. delta of 0.0001 for example, there is still a limited set of discrete values which are valid. Most measurements of physical quantities (typically with Analog to Digital converters) will be appropriate applications for this data type.

4.2.3 Containers

Though the arrays, matrices and the record structure are a good set of building blocks, the language provides a much richer set of containers to support a variety of ways of organizing the data. Line 22 in the specification creates a package foods_pkg through a process of known as instantiation of a generic package. In this case the resulting package supports another form of arrays namely Vector. Vectors with an index range of items_in_database_type and each item being of the type food_item_type is created by this instantiation. In a latter chapter we will develop our own generic package.

One of the parameters supplied in this instantiation is the operator or a function "=" for comparing two items. Of course, in this case it refers to the function declared in line 20.

Line 38 declares a variable of the data type Vector created above. The variable foods_database being private is not accessible to any users of this package. It is populated by loading the data from an external file by the function declared in line 39 also private.

Function Create_Meal declared in line 26 is however accessible to users of this package. As indicated, the vectors can be of any length to be specified by the user.

Once created the meal items can be updated - (indicated by in out), by the procedure AddItem.

4.2.4 Overloading

The package also declares 3 different procedures called show in lines 34, 35 and 40. Though they share the same name, each has a different argument list one of them with no arguments at all and the other 2 with distinct argument types. This technique called overloading can be extended to functions as well whereby functions can share the same name and argument list but differ in just the return type and still be distinct. In a way operators like "=" are the ultimate in overloading since the same symbol is used for the comparison of variables of any data type.

Listing 4.5: Foods package Specification

```
 1 : with Ada.Strings.Unbounded ; use Ada.Strings.Unbounded ;
 2 : with Ada.Containers.Vectors ;
 3 : with Ada.Text_IO ;
 4 :
 5 : package foods is
 6 :    verbose : boolean := false ;
 7 :    type carb_content_type is delta 0.1          -- gms
 8 :         range 0.0 .. 250.0 ;
 9 :    subtype glycemic_index_type is integer range 1..100 ;  -- Just a number
10 :    subtype calories_type is integer range 0..5000 ;       -- Calories or kJ
--
11 :    -- Each item's measurements are specified per serving
12 :    type food_item_type is
13 :       record
14 :          name : Unbounded_String ;              -- Name is not case sensitive
15 :          carbs : carb_content_type ;
16 :          calories : calories_type ;
17 :          glycemic_index : glycemic_index_type := glycemic_index_type'last ;
18 :       end record ;
19 :    type items_in_database_type is range 1..256 ;
20 :    function "="(Left, Right : food_item_type) return boolean ;
21 :
22 :    package foods_pkg is new Ada.Containers.Vectors( items_in_database_type , fo
23 :
24 :    -- Create a meal container. It is just an array of food items
25 :    -- except multiplied by the serving count/size
26 :    function Create_Meal( max_items : integer := 4 )
27 :                          return foods_pkg.Vector ;
28 :    procedure AddItem( meal : in out foods_pkg.Vector ;
29 :                       item_name : string ;
30 :                       servings : float := 1.0 ) ;
31 :    function Calories( meal : foods_pkg.Vector ) return calories_type ;
32 :    function Carbs( meal : foods_pkg.Vector ) return carb_content_type ;
33 :
34 :    procedure show ;
35 :    procedure show( meal : foods_pkg.Vector ) ;
```

```
36 :
37 : private
38 :    foods_database : foods_pkg.Vector ;
39 :    procedure Load (filename : string := "food_database.dat") ;
40 :    procedure show( item : food_item_type ) ;
41 : end foods ;
```

4.2.5 Separate files

The packages we have encountered so far had bodies in a single file. In large packages this may become impractical. The body of the foods package below illustrates how this could become more manageable. Line 57 declares the body of the procedure Load but indicates it is in a separate file.

Lines 4 and 5 of the Load procedure illustrates how the separate file is related to the package body.

4.2.6 Elaboration

- Lines 84 - 86 illustrate the notion of package elaboration. We can think of the package requiring initializations which take place during this process. In this case, in line 85 the procedure Load is invoked to load the foods database from the file.

4.2.7 Visitor pattern

Line 35 commences the implementation of Calories illustrating the significant advantage of using the Vector instead of arrays of our own definition. Line 42 applies the procedure Update_Calories to each element of the container meal. In this case the the procedure simply updates the total caloric value of each item. Line 37 indicates that a Cursor is supplied as an argument using which the element corresponding to that cursor can be retrieved as in line 39 for obtaining further details. All the housekeeping with respect to the indices and their limits and so on are handled by the Iterate procedure.

Listing 4.6: Foods package Body

```
 1 : with Ada.Text_Io ; use Ada.Text_Io ;
 2 : with Ada.Integer_Text_Io; Use Ada.Integer_Text_Io ;
 3 : package body foods is
 4 :
 5 :    function Create_Meal( max_items : integer := 4 )
 6 :                          return foods_pkg.Vector is
 7 :    begin
 8 :       return foods_pkg.To_Vector ( Ada.Containers.Count_Type(max_items) ) ;
 9 :    end Create_Meal ;
10 :
11 :    function "="(Left, Right : food_item_type) return boolean is
12 :    begin
13 :       return Left.name = Right.name ;
14 :    end "=" ;
15 :
16 :    procedure AddItem( meal : in out foods_pkg.Vector ;
17 :                               item_name : string ;
18 :                       servings : float := 1.0 ) is
19 :       use foods_pkg ;
20 :       ptr : foods_pkg.Cursor ;
```

```
21 :       item : food_item_type ;
22 :    begin
23 :       item.name := Trim(To_Unbounded_String(item_name),Ada.Strings.Right) ;
24 :       ptr := foods_pkg.find(foods_database , item) ;
25 :       if ptr = No_Element
26 :       then
27 :          raise Program_Error ;
28 :       end if ;
29 :       item := foods_pkg.Element(ptr) ;
30 :       item.calories := calories_type(servings) * item.calories ;
31 :       item.carbs := carb_content_type(servings) * item.carbs ;
32 :       foods_pkg.append( meal , item ) ;
33 :    end AddItem ;
34 :
35 :    function Calories( meal : foods_pkg.Vector ) return calories_type is
36 :       cals : calories_type := Calories_Type(0.0) ;
37 :       procedure Update_Calories( position : foods_pkg.Cursor ) is
38 :       begin
39 :          cals := cals + foods_pkg.Element( position ).calories ;
40 :       end Update_Calories ;
41 :    begin
42 :       foods_pkg.Iterate( meal , Update_Calories'Access ) ;
43 :       return cals ;
44 :    end Calories ;
45 :
46 :    function Carbs( meal : foods_pkg.Vector ) return Carb_Content_Type is
47 :       carbs : Carb_Content_type := Carb_Content_Type(0.0) ;
48 :       procedure Update_Carbs( position : foods_pkg.Cursor ) is
49 :       begin
50 :          carbs := carbs + foods_pkg.Element( position ).carbs ;
51 :       end Update_Carbs;
52 :    begin
53 :       foods_pkg.Iterate( meal , Update_Carbs'Access ) ;
54 :       return carbs ;
55 :    end Carbs ;
56 :
57 :    procedure Load(filename : string := "food_database.dat") is separate ;
58 :
59 :    package carbs_io is new Ada.Text_IO.Fixed_IO( Carb_Content_Type ) ;
60 :    use carbs_io ;
61 :
62 :    procedure show(item : food_item_type) is
63 :    begin
64 :       put( to_string(item.name) ) ;
65 :       Set_Col( 35 ) ;
66 :       put( item.carbs ) ;
67 :       Set_Col(45) ;
68 :       put(item.calories) ;
69 :       new_line ;
70 :    end show ;
71 :
72 :    procedure show is
73 :    begin
74 :       Put("No_of_Items_in_the_foods_database");
75 :       Put( Integer(Foods_Pkg.Length( Foods_database )) ) ;
```

```
76 :         New_Line ;
77 :         show( foods_database ) ;
78 :      end show ;
79 :      procedure show( meal : foods_pkg.Vector )  is
80 :         procedure show( position : foods_pkg.Cursor ) is
81 :         begin
82 :            show(foods_pkg.element( position )) ;
83 :         end show ;
84 :      begin
85 :         Put("Food_Item");
86 :         Set_Col(37);
87 :         Put("Carbs");
88 :         Set_Col(50);
89 :         Put("Calories");
90 :         New_Line ;
91 :         foods_pkg.Iterate( meal , show'Access ) ;
92 :      end show ;
93 : begin
94 :      Load ;
95 : end foods ;
```

4.2.8 Package Elaboration

Line 93 of the package body introduces the concept of the package elaboration. The fragment starting with begin till the end is part of the elaboration code - the initialization of the package. This fragment will be executed only once in an executable before the main program begins. In this example the load procedure is called to populate the foods database.

4.2.9 Source code organization

Line 57 of the body introduces the keyword separate which indicates that the implementation of the load procedure is in a separate file as shown below. In packages which have too many procedures or functions, it may be appropriate to provide separate files. In this case considering the lines of code to implement the load function, being separate improves readability.

Listing 4.7: Separate procedure Load

```
 1 : with Ada.Text_Io ; use Ada.Text_Io ;
 2 : with Ada.Integer_Text_Io; Use Ada.Integer_Text_Io ;
 3 :
 4 : separate ( foods )
 5 : procedure Load(filename : string := "food_database.dat") is
 6 :    use Ada.Strings.Unbounded ;
 7 :    food_db : file_type ;
 8 :
 9 :    item_name : string(1..30) ;
10 :    item_name_length : natural ;
11 :
12 :    package carbs_io is new Ada.Text_IO.Fixed_IO( Carb_Content_Type ) ;
13 :    use carbs_io ;
14 :
15 :    Char : Character ;
16 :    End_Of_Name : constant Character := ':' ;
17 :    Food_Item : Food_Item_Type ;
18 :    tempint : integer ;
19 : begin
```

```
20 :     open( food_db , in_file , filename ) ;
21 :     while not end_of_file(food_db)
22 :     loop
23 :        --
24 :        -- First read the food item name which is terminated
25 :        -- by a special character. Thus can be of a variable length
26 :        Item_Name_Length := 0;
27 :        while not End_Of_File(Food_Db)
28 :        loop
29 :           Get(Food_Db,Char) ;
30 :           if Char = End_Of_Name
31 :           then
32 :              exit ;
33 :           end if ;
34 :           Item_Name_Length := Item_Name_Length + 1 ;
35 :           if Item_Name_Length <= Item_Name'Length
36 :           then
37 :              Item_Name(Item_Name_Length) := Char ;
38 :           end if ;
39 :        end loop ;
40 :        Food_Item.Name := Trim(
41 :                             To_Unbounded_String(Item_Name(1..Item_Name_Length))
42 :                          , Ada.Strings.Right) ;
43 :        -- Now we can read the rest of the items
44 :        -- each just separated by a space
45 :        Get( Food_Db , Food_Item.Carbs ) ;
46 :
47 :        Get( Food_Db , tempint ) ;
48 :        food_item.calories := tempint ;
49 :
50 :         Get( Food_Db , tempint ) ;
51 :        food_item.glycemic_index := tempint ;
52 :
53 :        -- An item has been assembled
54 :        -- Add it to the database
55 :        Foods_Pkg.Append( Foods_Database , Food_Item ) ;
56 :        if verbose
57 :        then
58 :           show( food_item ) ;
59 :        end if ;
60 :     end loop ;
61 :     close( food_db ) ;
62 : end Load ;
```

Line 4 in the above listing specifies that this procedure is the part of the package foods. Lines 12 and 13 instantiate the Fixed_Io package for the type Carb_Content_Type which is defined to be of fixed point datatype.

Also to note is the use clause in line 6 due to which it is not necessary to qualify the references to Trim in line 40 or even the implicit conversion of Item_Name to the Unbouned_String datatype. In general most programming standards may dissuade the use of the use clause but for language defined library packages, the use clauses may in fact improve readability.

4.2.10 Attributes

The program foodsdb listed below is a simplistic user of the package discussed above. Starting at line 10 the attributes of the data types declared and objects of those data types are printed out. Line 29 for example displays the offset of the field Carbs within the record using the Position attribute - in terms of Storage_Units.

Listing 4.8: Foods Database user foodsdb

```
 1 : with Ada.Text_IO;              use Ada.Text_IO;
 2 : with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
 3 : with foods;
 4 : procedure foodsdb is
 5 :    Cc  : foods.carb_content_type;
 6 :    Fi  : foods.food_item_type;
 7 :    Idb : foods.items_in_database_type;
 8 : begin
 9 :    Put_Line ("------------------------------------------------");
10 :    Put ("Carb_Content_Type'Size");
11 :    Set_Col (30);
12 :    Put (foods.carb_content_type'Size);
13 :    New_Line;
14 :    Put ("Carb_Content_Object_Size");
15 :    Set_Col (30);
16 :    Put (Cc'Size);
17 :    New_Line;
18 :
19 :    Put ("Food_Item_Type'Size");
20 :    Set_Col (30);
21 :    Put (foods.food_item_type'Size);
22 :    New_Line;
23 :    Put ("Food_Item_object_Size");
24 :    Set_Col (30);
25 :    Put (Fi'Size);
26 :    New_Line;
27 :    Put ("fi.Carbs_Position");
28 :    Set_Col (30);
29 :    Put (Fi.carbs'Position);
30 :    New_Line;
31 :    Put ("fi.Calories_Position");
32 :    Set_Col (30);
33 :    Put (Fi.calories'Position);
34 :    New_Line;
35 :    Put ("fi.glycemic_Index_position");
36 :    Set_Col (30);
37 :    Put (Fi.glycemic_index'Position);
38 :    New_Line;
39 :    Put ("Items_In_Database_type'Size");
40 :    Set_Col (30);
41 :    Put (foods.items_in_database_type'Size);
42 :    New_Line;
43 :    Put ("Items_In_Database_Object_Size");
44 :    Set_Col (30);
45 :    Put (Idb'Size);
46 :    New_Line;
47 :    Put_Line ("------------------------------------------------");
```

```
48 :
49 :     foods.show;
50 : end foodsdb;
```

The output below clarifies that Carb_Content_Type requires 12 bits though since the storage_unit on my system is 8 bits in length, any objects of that type occupy 16 bits. Similarly the entire record type requires 224 bits though the objects of that type occupy 256 bits.

Listing 4.9: foodsdb output

```
Carb_Content_Type'Size                    12
Carb_Content Object Size                  16
Food_Item_Type'Size                      256
Food_Item object Size                    256
fi.Carbs Position                         16
fi.Calories Position                      20
fi.glycemic_Index position                24
Items_In_Database_type'Size                9
Items_In_Database Object Size             16

No of Items in the foods database          11
Food Item                        Carbs        Calories
Apple                            22.0              80
Avocado                           3.0              60
Grapefruit                       16.0              60
Pomegranate                      26.0             110
Dark Chocolate                   56.7             545
Almond Honey                     48.0             558
White Chocolate                  55.9             562
Thai Stir-Fry                    45.0             310
Veggie Burger                    15.0             110
Burrito                          50.0             300
Teriyaki Wrap                    51.0             310
```

4.2.11 Exercises

Expand the food database to contain the protein and fat content as well.

Daily Meal A person enjoys several meals a day. Add an appropriate support to maintain a daily meal. Provide an end of day procedure that provides a summary of calories, proteins and carbs consumed over the day.

Spurious duplicates Study the complete specification of the predefined package Ada.Strings.Unbounded which is used in this package. Modify the package so that blanks in food item names and their case are not significant. For example “Orange Juice”, “OrangeJuice” and ”ORANGEJUICE” should be treated the same while adding/modifying items to a meal as well as while search for items.

Faster food database The foods_database is maintained in the same order in which it is in the input file. Finding items in this might take a long time - particularly as the number of items in the database grows the time required to add a new item to the meal is going to be higher. Investigate other language defined containers which might help with this problem.

Chapter 5

Generics

We had a brief encounter with generic packages in an earlier chapter. In this chapter a generic package part of the predefined library framework is explored in some detail. Building upon this we will develop our own generic package.

5.1 Projectlet constants

The language defined environment includes a rich set of containers which are designed as generic packages. These are designed to be instantiated to contain objects of userś design. In this projectlet we build a facility to maintain a table of physical constants with floating point values. Once populated, we expect to lookup different constants by name.

5.1.1 Instantiation

For example the specification of the package constants defines a data type Constant_Type starting at line 6. This record structure contains several fields. Lines 17 - 21 instantiate a library package Ordered_Maps for this data type. As indicated in line 19, the key for this table is of the type Unbounded_string implying the key could be of varying lengths. Elements of this table as specified in line 20 is the Constant_Type declared earlier.

The specification of the package Ordered_Maps indicates it is a random access data structure implying we can lookup a constant based on a key. However the table is also ordered so there is an inherent traversal order. The package makes no assumptions about either the key or the data elements and hence expects the user to provide functions for < for the key and = for the elements. Since the constants_table uses an unbounded_string for its keys, the predefined function = defined for the data type unbounded_string is provided as a parameter.

With the ability to provide our own functions for < for determining the key ordering and for = to support searching for values, the package provides a lot of flexibility.

Listing 5.1: Constants specification

```
1 : with Ada.Strings.Unbounded ; use Ada.Strings.Unbounded ;
2 : with Ada.Containers.Ordered_Maps ;
3 :
4 : package Constants is
5 :
6 :    type Constant_Type is
7 :    record
8 :       Name : Unbounded_String ;
9 :       Description : Unbounded_String := Null_Unbounded_String ;
```

```
10 :       Units : Unbounded_String := Null_Unbounded_String ;
11 :       Value : Float ;
12 :    end record ;
13 :
14 :    function Equal( Left, Right : Constant_Type ) return Boolean ;
15 :    function LessThan( Left, Right : Constant_Type ) return Boolean ;
16 :
17 :    package Constants_Table_Pkg is new Ada.Containers.Ordered_Maps
18 :      (
19 :       Key_Type => Unbounded_String ,
20 :       Element_Type => Constant_Type ,
21 :       "=" => Equal ,
22 :       "<" => Ada.Strings.Unbounded."<"
23 :      ) ;
24 :
25 :    type Constants_Table_Type is new Constants_Table_Pkg.Map with null record ;
26 :
27 :    function Create( Filename : String ) return Constants_Table_Type ;
28 :    procedure Save( Table : Constants_Table_Type ;
29 :              Filename : String ) ;
30 :    procedure Load( Table : in out Constants_Table_Type ;
31 :              Filename : String ) ;
32 :
```

5.1.2 Operating on the container

The instantiation creates a custom package Constants_Table_Package in line 17 of the spec that supports a container for objects of the type Constant_Type. As illustrated in line 38 of the body, we can iterate through each object in a container ie. a constants table. Iteration involves applying a procedure Show to each of the elements in the table.

The procedure Show defined in line 199 illustrates the notion of a Cursor which points to each entry in the container.

New entries are inserted using the Insert procedure as in line 96. Line 89 illustrates looking up the table for a constant.

Listing 5.2: Constants body

```
1 : with Ada.Text_Io; use Ada.Text_Io ;
2 : with Ada.Float_Text_Io; use Ada.Float_Text_Io ;
3 : with GNAT.Regpat ; use GNAT.Regpat ;
4 :
5 : package body Constants is
6 :
7 :    function Create( Filename : String ) return Constants_Table_Type is
8 :       Table : Constants_Table_Type ;
9 :    begin
10 :      return Table ;
11 :   end Create ;
12 :
13 :   procedure Save( File : Ada.Text_Io.File_Type ;
14 :             Const : access Constant_Type ) is
15 :   begin
16 :      Put( File , To_String(Const.Name) ) ;
```

```
17 :       Put( File , ":" ) ;
18 :       Put( File , To_String(Const.Description) ) ;
19 :       Put( File , ":" ) ;
20 :       Put( File , To_String(Const.Units) ) ;
21 :       Put( File , ":" ) ;
22 :       Put( File , Const.Value ) ;
23 :       New_Line( File ) ;
24 :    end Save ;
25 :
26 :    SavingFile : File_Type ;
27 :    procedure Save( Position : Constants_Table_Pkg.Cursor ) is
28 :       Const : aliased Constant_Type ;
29 :    begin
30 :       Const := Constants_Table_Pkg.Element( Position ) ;
31 :       Save( SavingFile , Const'Access ) ;
32 :    end Save ;
33 :
34 :    procedure Save( Table : Constants_Table_TYpe ;
35 :              Filename : String ) is
36 :    begin
37 :       Create(SavingFile,Out_File,Filename) ;
38 :       Constants_Table_Pkg.Iterate( Constants_Table_Pkg.Map( Table ) , Save'Acce
39 :       Close(SavingFile) ;
40 :    end Save ;
41 :
42 :    Commentlinepat : Pattern_Matcher := Compile ("\w*\#.*");
43 :    Constantpat    : Pattern_Matcher := Compile( "(\w+)\s*\:\s*(.*)\s*\:\s*(.+)\
44 :    procedure Load( Table : in out Constants_Table_Type ;
45 :              Filename : String ) is
46 :       Constfile : File_Type ;
47 :       Line : String (1..132) ;
48 :       Linelen : Natural ;
49 :       procedure ProcessLine( Inpline : String ) is
50 :   Matched : Match_Array (0 .. 4 );
51 :   Const : Constant_Type ;
52 :       begin
53 :   if Match( Commentlinepat , Inpline )
54 :   then
55 :      Put("Found Comment : ");
56 :      Put_Line( Inpline ) ;
57 :      return ;
58 :   end if ;
59 :   Match( Constantpat , Inpline , matched) ;
60 :   if Matched(1) /= No_match
61 :   then
62 :      Put("Name : "); Put_Line( Inpline( Matched(1).First .. Matched(1).Last ) )
63 :      Put("Desc : "); Put_Line( Inpline( Matched(2).First .. Matched(2).Last ) )
64 :      Put("Units: "); Put_Line( Inpline( Matched(3).First .. Matched(3).Last ) )
65 :      Put("Value: "); Put_Line( Inpline( Matched(4).First .. Matched(4).Last ) )
66 :      Const.Name := To_Unbounded_String(Inpline( Matched(1).First .. Matched(1).
67 :      Const.Description := To_Unbounded_String(Inpline( Matched(2).First .. Matc
68 :      Const.Units := To_Unbounded_String(Inpline( Matched(3).First .. Matched(3)
69 :      Const.Value := Float'Value(Inpline( Matched(4).First .. Matched(4).Last ))
70 :      Constants_Table_Pkg.Insert( Constants_Table_Pkg.Map(Table) , Key => Const.
71 :   end if ;
```

```
 72 :       end ProcessLine ;
 73 :    begin
 74 :       Put("Loading constants from "); Put_Line(Filename) ;
 75 :       Open(Constfile,In_File,Filename) ;
 76 :       while not End_Of_File(Constfile)
 77 :       loop
 78 :    Get_Line(Constfile, Line , Linelen) ;
 79 :    ProcessLine( Line(1..Linelen) ) ;
 80 :       end loop ;
 81 :       Close(Constfile) ;
 82 :    end Load ;
 83 :
 84 :    function Get( Table : Constants_Table_Type ;
 85 :           Name : String )
 86 :         return Constant_Type is
 87 :       Const : Constant_Type ;
 88 :    begin
 89 :       Const := Constants_Table_Pkg.Element( Constants_Table_Pkg.Map(Table) , To
 90 :       return Const ;
 91 :    end Get ;
 92 :
 93 :    procedure Add( Table : in out Constants_Table_Type ;
 94 :            Const : Constant_Type ) is
 95 :    begin
 96 :       Constants_Table_Pkg.Insert( Constants_Table_Pkg.Map(Table) ,
 97 :                                   Key => Const.Name ,
 98 :                                   New_Item => Const ) ;
 99 :    end Add ;
100 :
101 :    procedure Update( Table : in out Constants_Table_Type ;
102 :              Const : Constant_Type ) is
103 :    begin
104 :       null ;
105 :    end Update ;
106 :
107 :    procedure Show( Const : Constant_Type ) is
108 :    begin
109 :       Put( To_String(Const.Name) ) ;
110 :       Set_Col( 10 ) ;
111 :       Put( To_String(Const.Description) ) ;
112 :       Set_Col( 40 );
113 :       Put( To_String(Const.Units) ) ;
114 :       Set_Col( 60 ) ;
115 :       Put( Const.Value ) ;
116 :       New_Line ;
117 :    end Show ;
118 :
119 :    procedure Show( Position : Constants_Table_Pkg.Cursor ) is
120 :       Const : Constant_Type ;
121 :    begin
122 :       Const := Constants_Table_Pkg.Element( Position ) ;
123 :       Show( Const ) ;
124 :    end Show ;
125 :
126 :
```

```
127 :     procedure Show( Table : Constants_Table_Type ) is
128 :     begin
129 :        Constants_Table_Pkg.Iterate( Constants_Table_Pkg.Map(Table) , Show'Access
130 :     end Show ;
131 :
132 :
133 :     function Equal( Left, Right : Constant_Type ) return Boolean is
134 :     begin
135 :        return Ada.Strings.Unbounded."="( Left.Name , Right.Name ) ;
136 :     end Equal ;
137 :
138 :     function LessThan( Left, Right : Constant_Type ) return Boolean is
139 :     begin
140 :        return Ada.Strings.Unbounded."<"( Left.Name , Right.Name ) ;
141 :     end LessThan ;
142 :
143 : end Constants ;
```

5.2 Projectlet integrate

This projectlet develops a package to integrate any function using the Newton Coates and in particular the Simpsons rule. This implementation will illustrate two extremely elegant and powerful features of Ada - the generics and function or procedure access. In the chapter on data structures, we had a brief encounter with the procedure access in the Visitor pattern.

5.2.1 Generic Package Specification

The specification for the package integrate is similar to any other package specification encountered earlier except for the instantiation parameters as specified in line 2 below. The package is parametrized by a data type the only condition being that it is a floating point data type - as indicated by digits <>. Thus the Real_T could be float or long_float or long_long_float or any other restrictive type derived from any of these.

The package itself is only "generic" meaning this cannot be used directly till a real package is created by instantiating it with a valid data type.

Function access Lines 4 and 5 of the specification declare a new data type for referring to the integrand. Each of the integrators accept an integrand argument.

Listing 5.3: Package Specification

```
 1 : generic
 2 :    type Real_T is digits <> ;
 3 : package Integrate is
 4 :    type Integrand_Type is access
 5 :      function ( X : Real_T ) return Real_T ;
 6 :    function Newton_Coates_3( X1, X2 : Real_T ;
 7 :                Integrand : Integrand_Type ) return Real_T ;
 8 :    function Newton_Coates_4( X1, X2 : Real_T ;
 9 :                Integrand : Integrand_Type ) return Real_T ;
10 :    function Newton_Coates_5( X1, X2 : Real_T ;
11 :                Integrand : Integrand_Type ) return Real_T ;
12 :
13 :    function Simpsons( X1, X2 : Real_T ;
14 :                Integrand : Integrand_Type ) return Real_T
15 :      renames Newton_Coates_3 ;
```

```
16 :
17 : end Integrate ;
```

The instantiation below creates a package integrate that can then take float arguments.

Listing 5.4: Package Instantiation

```
1 : with Integrate ;
2 :
3 : package Float_Integrate is new Integrate ( Float ) ;
```

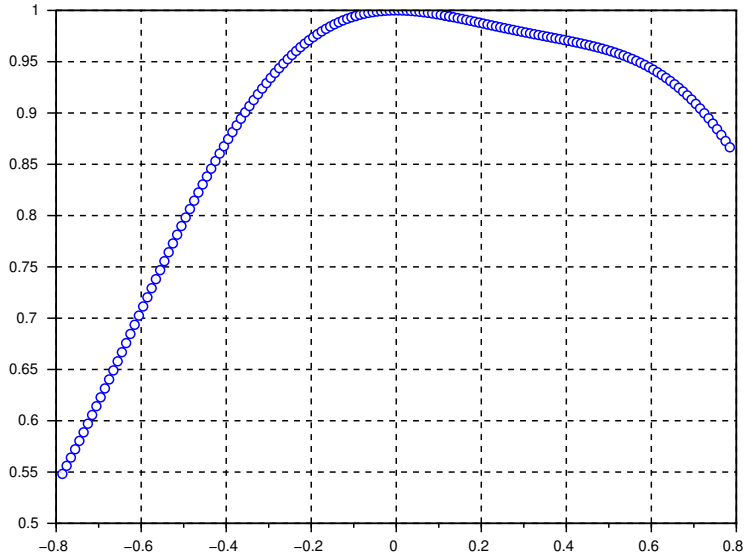
5.2.2 Package user

As recommended in the chapter on unit tests we develop the unit test framework for the integrator package. The unit test illustrates the application of the package in lines 37 - 39. Essentially the function f1 is integrated from -$\pi/4$ to $\pi/4$. The function itself is plotted as shown. The analytical solution of this integration can be found to be $\sqrt{2}$. The assertion in lines 45 - 46 confirms that the integrator does indeed yield a value within a margin of 0.0001.

Listing 5.5: Unit test for the integrator

```
 1 :
 2 : with Ada.Text_Io ; use Ada.Text_Io ;
 3 : with Ada.Complex_Text_Io ;
 4 : with Ada.Float_Text_Io ; use Ada.Float_Text_Io ;
 5 : with Ada.Numerics.Elementary_Functions ;
 6 :      use Ada.Numerics.Elementary_Functions ;
 7 :
 8 : with AUnit.Assertions ; use AUnit.Assertions ;
 9 : with float_integrate ;
10 :
11 : package body integrate_tests is  -- [template/$]
12 :
13 :    procedure Register_Tests( T : in out integrate_Test ) is
14 :        use AUnit.Test_Cases.Registration ;
15 :     begin
16 :        Register_Routine( T , Test_integrate'access , "integrate Test") ;
17 :     end Register_Tests ;
18 :
19 :     function Name( T : integrate_Test ) return Message_String is
20 :     begin
21 :         return Format("integrate Tests") ;
22 :     end Name ;
23 :
24 :     function f1( t : float ) return float is
25 :        f : float ;
26 :     begin
27 :         f := cos(t) +
28 :              sqrt( 1.0 + t**2) *
29 :                 sin(t) ** 3 *
30 :                 cos(t) ** 3 ;
31 :         return f;
32 :     end f1 ;
33 :
34 :     procedure Test_integrate( T : in out Test_Cases.Test_Case'Class ) is
35 :        Y : float ;
36 :        EPSILON : constant float := 0.1e-3 ;
```

Figure 5.1: Plot of $\cos(x) + sqrt1 + x^2 * sin^3(x) * cos^3(x)$




```
37 :      begin
38 :         Y := float_integrate.Newton_Coates_5(
39 :                                                 - Ada.numerics.pi / 4.0 ,
40 :                                                   Ada.numerics.pi / 4.0 ,
41 :                                                   f1'access ) ;
42 :         -- Put(Y) ;
43 :         -- Put(Ada.Numerics.Elementary_Functions.sqrt(2.0)) ;
44 :         -- new_line ;
45 :
46 :         Assert( abs((Y - sqrt(2.0))) < EPSILON ,
47 :                 "Integral of function is sqrt(2.0)");
48 :
49 :      end test_integrate ;
50 :
51 : end integrate_tests ;
```

5.2.3 Generic package body

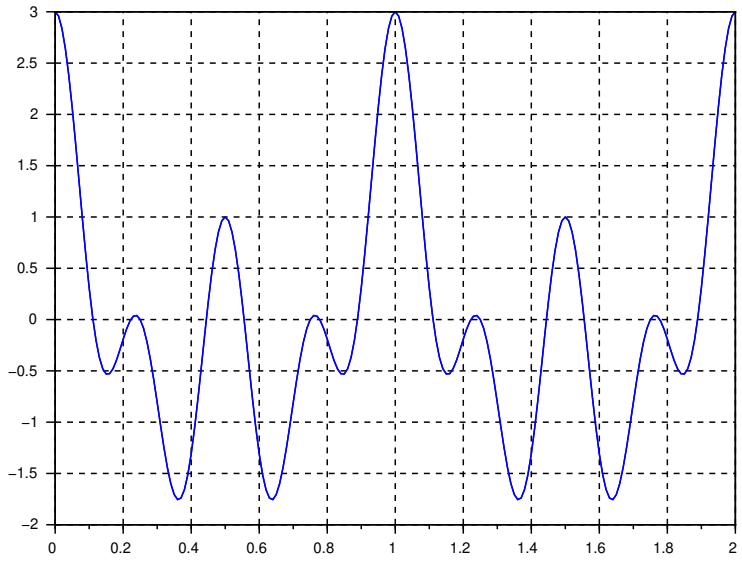
The body of a generic body does not seem any different from any other package body. No assumptions are made about the data type Real_T so all local temporary variables are declared as Real_T as in lines 5 - 11. The impelementation does not also know anything about the integrand except that it is a function returning a value of the same data type Real_T as its argument - in other words having the signature specified in line 5 of the package specification.

The integrand is invoked in lines 13 - 15 for different values and finally the result is estimated in line 17.

Listing 5.6: Package Instantiation

```
1 : with Integrate ;
2 :
```

Figure 5.2: Plot of sum of the harmonics of cos(2*pi*freq*t)




```
3 : package Float_Integrate is new Integrate ( Float ) ;
```

5.3 Exercises

Integration formulae Even using the order 5 the Newton Coates formulae are somewhat coarse. In order to improve the accuracy, we could divide the user specified range into a number of smaller intervals and apply the formulae. Implement a Newton Coates formula of the order 5 accepting a number of sub divisions as an argument. Evaluate the performance for some cyclic functions.

Harmonics Add a unit test to confirm that the integration of

cos(2*pi*t) + cos(2*pi*2*t) + cos(2*pi*4*t)

will yield 0 over an integral number of cycles (e.g. from 0 to 2). Does the unit test confirm the result? If not why not?

Fixed point generics The data type Carb_Content_Type defined in the package foods in an earlier chapter is a non integer data type but not floating point; it is a fixed point data type. Implement the body and a unit test for a univariate polynomial package with the following specification:

```
generic
   type numtype is delta <> ;
package polynomial is
    type Polynomial_Type is array (natural range <>)
         of num_type ;
    -- Create a polynomial of the specified order
```

```
    -- and initialize all coefficients to the value default
    function Create( order : integer ;
                     default : num_type ) return Polynomial_Type ;
    procedure Set( pol : in out polynomial_type ;
                   index : integer ;
                   coefficient ) ;
    procedure Load( filename : string ;
                    pol : out polynomial_type ) ;
    procedure Save( filename : string ;
                    pol : polynomial_type ) ;
    function Evaluate( pol : polynomial_type ;
                       variable : num_type )
                    return num_type ) ;
end polynomial ;
```

Appendices

Appendix A

Code Examples

This book and all the code examples can be cloned from

```
https://gitlab.com/RajaSrinivasan/TechAdaBook.git.
```